SYSTEM AND METHOD FOR DERIVING A PROCESS-BASED SPECIFICATION

[0001] This application for patent hereby claims priority to U.S. Provisional Patent Application Serial No. 60/533,376 entitled "System and Method for deriving a Process-Based Specification" by Hinchey et al., which was filed on December 22, 2003. This provisional patent application is hereby incorporated by reference.

ORIGIN OF THE INVENTION

[0002] The invention described herein was made by employees of the United States Government, and may be implemented or manufactured and used by or for the Government for governmental purposes without the payment of any royalties thereon or therefor.

FIELD OF THE INVENTION

[0003] The present invention relates to computer and software engineering environments, and more particularly, the present invention relates to software development, requirements definition, formal methods, system validation and verification, and code generation, both automatic and manual. Additionally, the herein disclosed invention relates to the fields of chemical or biological process design or mechanical system design, and, generally to any field where the behaviors exhibited by a process to be designed is described by means of a set of scenarios expressed in natural language, or some appropriate graphical notation.

BACKGROUND OF THE INVENTION

[0004] Complex (software and hardware) systems are developed for numerous applications and processes, including the automated control of spacecraft operations and ground systems. Complex

software and hardware systems, however, may encounter problems. The cause of potential faults and defects, such as redundancies, deadlocks, and omissions, may be difficult to determine, especially when the system is distributed and has parallel execution paths. Formal specification methods provide a means for avoiding or discovering such defects. Currently available techniques to formally specify software and hardware, however, can be difficult and time consuming to use.

[0005] Conventional processes for (hardware and software) system development include code generation (either automated or manual) from a specification that includes a specification language along with a tool kit. These processes enable model checking, verification, and automatic code generation. Disadvantages with these approaches include the user specifying every low-level detail of the system in advance. Thus, a system specification might be difficult to develop, understand, and modify. Further, difficulties may exist in establishing that the resulting code represents the customer's requirements, because the requirements are in natural language, and not in a specification language that is amenable to analysis.

[0006] Other conventional approaches include state-based approaches, employing, for example, statecharts or use-cases. These approaches may not offer the capability to check for errors, deadlocks, omissions, and the like, which formal specification languages provide, unless additional constraints are added. These constraints can be unwieldy or introduce inefficiencies into the development process, or indeed result in the incorrect system being developed. The same difficulties described above (i.e., the difficulty of establishing that the resulting code represents the customer's requirements) also apply to these other conventional approaches.

SUMMARY OF THE INVENTION

[0007] Accordingly, the disclosed embodiments of the present invention are directed toward system and methods for deriving a process-based specification that solves or reduces the problems within the conventional art. According to the disclosed embodiments of the present invention, a method for deriving a process-based specification for a system is disclosed. The method includes deriving a trace-based specification from a non-empty set of traces. The method includes mathematically inferring that the process-based specification is mathematically equivalent to the trace-based specification. Accordingly, the disclosed embodiments of the present invention are directed toward said methods for deriving a process-based specification that solves or reduces the problems within the conventional art.

[0008] Additional features or advantages of the disclosed embodiments are set forth in the description that follows, and in part will be implied from the description, or may be learned by practice of the invention. The objectives and other advantages of the invention are realized and obtained by the structure and methods particularly pointed out in the written description and the claims as well as the appended drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

[0009] The accompanying drawings, which are included to provide further understanding of the disclosed invention, illustrate embodiments of the invention, and together with the descriptions serve to explain the principles of the invention. In the drawings:

[0010] Figure 1 illustrates a software development system according to the disclosed embodiments.

[0011] Figure 2 illustrates a flowchart for deriving formal specifications and code from scenarios according to the disclosed embodiments.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0012] Aspects of the invention are disclosed in the accompanying description. Alternate embodiments of the present invention may be derived without parting from the spirit or scope of the present invention. It should be noted that like elements in the figures are indicated by like reference numbers.

[0013] Figure 1 illustrates a software development system 100 according to the disclosed embodiments. Software development system 100 includes a data flow and processing points for the data. Software development system 100 is representative of (i) computer applications and electrical engineering applications such as chip design and other electrical circuit design, (ii) business management applications in areas such as workflow analysis, (iii) artificial intelligence applications in areas such as knowledge-based systems and agent-based systems, (iv) highly parallel and highly-distributed applications involving computer command and control and computer-based monitoring, and (v) any other area involving process, sequence or algorithm design. According to the disclosed embodiments, software development system 100 mechanically converts different types of specifications (either natural language scenarios or descriptions, or trace specifications, which are effectively pre-processed scenarios) into process-based formal specifications on which model checking and other mathematics-based verifications are performed, and then optionally converts the formal specification into code.

[0014] Software development system 100 receives natural language scenarios 110. A scenario is natural language text that describes the software's actions in response to incoming data and the internal goals of the software. Scenarios also may describe communication protocols between systems and between the components within the systems. Scenarios also may be known as use-cases. A scenario describes one or more potential executions of a system, describing what happens in a particular situation, and what range of behaviors is expected from or omitted by the system under various conditions.

[0015] The set of natural language scenarios 110 is constructed in terms of individual scenarios written in a structured natural language. Different scenarios may be written by different stakeholders of the system, corresponding to the different views they have of how the system will perform. Natural language scenarios 110 may be generated by a user with or without mechanical or computer aid. The set of natural language scenarios 110 provides the descriptions of actions that occur as the software executes. Some of these actions will be explicit and required, while others may be due to errors arising, or as a result of adapting to changing conditions as the system executes.

[0016] For example, if the system involves commanding space satellites, scenarios for that system may include sending commands to the satellites and processing data received in response to the commands. Natural language scenarios 110 should be specific to the technology or application domain to which it is applied. A fully automated general purpose approach covering all domains is technically prohibitive to implement in a way that is both complete and consistent. To ensure consistency, the domain of application must be specific-purpose. For example,

scenarios for satellite systems may not be applicable as scenarios for systems that manufacture agricultural chemicals.

[0017] Natural language scenarios 110 are input to a context sensitive editor 120. Context sensitive editor 120 also has access to a database of domain parameters 115. Domain parameters reside in a database of parameters 115 that describes the terms usable and allowable in natural language scenarios 110, that is, the "context". The parameters allow for a wide range of domains and applications to be used with specialized terms in the scenarios. Different vocabulary applies to different areas of technology, and the vocabulary may change for different applications and different domains. Database of domain parameters 115 may comprise any terms or language that may be defined using grammar. Database of parameters 115 also will contain a dictionary of application-related events and actions.

[0018] Context sensitive editor 120 takes natural language scenarios 110 and database of domain parameters 115, and produces a trace specification 130 based on the inputted scenarios. Trace specification 130 correlates to natural language scenarios 110. In formal methods, and other areas of software engineering, a trace is a sequence of events that a process has engaged in up to a given point in time. The set of all possible behavior patterns of a process is represented by the set of traces of that process. Context sensitive editor 120 produces trace specification 130 from natural language scenarios 110. Context sensitive editor 120 may perform these actions automatically.

[0019] Trace specification 130 includes traces that are a list of all possible orderings of actions as described by the developers that are taking place within the scenarios. More specifically, traces are lists of computational actions. Traces

list the actions in a sequential manner. In other words, the traces of trace specification 130 may not include conditional statements or actions. Conditions in the input scenarios 110 are maintained in the set of traces 130 by introducing multiple traces describing alternate execution patterns. As a result, the number of traces in the set of traces 130 is likely to exceed the number of scenarios in the set of natural language scenarios 110. Traces also may be known as sequences, events, or event-traces.

[0020] Traces also may be known as a sequence of steps. Trace specification, or set of traces, 130 incorporates a mathematical language that is behavioral. The set of traces 130 describes all possible behavior patterns of the system as described by the developers and/or users in the natural language scenarios 110. From this behavioral description, the disclosed embodiments can mathematically infer a more general description from specific descriptions. Specific execution paths within natural language scenarios 110 may be input, and generalized executions generated from the specific executions. The more generalized description is amenable to analysis and enables the highlighting of erroneous or problematic execution paths.

[0021] Inference Engine 150 is a theorem prover, also known as an automatice theorem prover, in which laws of concurrency 145 have been embedded. Laws of concurrency 145 are rules detailing equivalences between sets of processes combined in various ways, and/or relating process-based descriptions of systems or system components to equivalent sets of traces. The embedding may be "shallow" in which case the theorem prover operates on syntactic equivalences. The embedding may be "deep", in which case the theorem prover operates at the level of semantic equivalence.

The latter is preferred, but the disclosed embodiments also may operate at a shallow level.

[0022] The embedding of laws of concurrency 145 in inference engine 150 may be validated by using the embedding to prove the laws of concurrency, which are known to be correct. An example of the laws of concurrency 145 are given in "Concurrent Systems: Formal Development in CSP" by M.G. Hinchey and S.A. Jarvis, McGraw-Hill International Series in Software Engineering, New York and London, 1995, herein incorporated by reference.

[0023] Laws of concurrency 145 may be expressed in any suitable language for describing concurrency. These languages include, but are not limited to, CSP (Communicating Sequential Processes), CCS (Calculus of Communicating Systems), and variants of these languages. The theorem prover forming the basis of inference engine 150 may be any available conventional commercial or academic theorem prover, or a bespoke theorem prover.

[0024] Inference engine 150 involves a deep embedding of the laws of concurrency expressed in CSP in the freely-available theorem prover ACL2. Other embeddings of laws of concurrency (or equivalence of combinations of processes) expressed in other languages and implemented in other theorem provers also embody the disclosures described herein, and may be substituted for inference engine 150 and are equivalent to the disclosed embodiments of the present invention.

[0025] Inference engine 150 takes as input trace specification 130, and combined with its embedding of the laws of concurrency 145, reverses the laws of concurrency 145 to infer a process-based specification 160. The process-based specification 160 is mathematically and provably equivalent to the trace-based

specification 130. Mathematically equivalent does not necessarily mean mathematically equal. Mathematical equivalence of A and B means that A implies B and B implies A. Note that applying the laws of concurrency 145 to the process-based specification 160 would allow for the retrieval of a trace-based specification that is equivalent to the trace-based specification 130. Note that the process-based specification is mathematically equivalent to rather than necessarily equal to the original trace-based specification 130. This feature indicates the process may be reversed, allowing for reverse engineering of existing systems, or for iterative development of more complex systems.

[0026] The mechanism by which the inference engine 150 will infer the process-based specification 160 will vary depending on the theorem prover that is used as the basis of the inference engine 150. This feature generally involves a combination of rule rewriting and searching a tree of possible rewritings to determine which process combinations correspond to the traces in the trace-based specification 130. The choice of theorem prover used as the basis of the inference engine 160 will, combined with the completeness of the embedding of the laws of concurrency 145, determine the performance of the system.

[0027] Formal specification analyzer 170 receives as input the process-based formal specification 160. Formal specification analyzer 170 allows the user to manipulate the formal specification 160 in various ways. Alternate implementations, on different hardware and software architectures, may be considered. The formal specification analyzer 170 allows the user to examine the system described by the natural language scenarios 110, and to manipulate it so as to execute on a different combination of (local or distributed) processors. The

process-based formal specification 160 may be analyzed to highlight undesirable behavior, such as race conditions, and equally important, to point out errors of omission in the original natural language scenarios 110 and/or the trace specification 130. The formal specification analyzer 170 is an optional but useful stage in the disclosed embodiments of the present invention. If the formal specification analyzer 170 is not used, then the process-based specification 160 and the revised formal specification 175 are identical. Hence, if the formal specification analyzer 170 is not used then all references to the revised formal specification 175 disclosed below also apply to the process-based specification 160.

[0028] Revised formal specification 175 highlights behavior patterns that were previously not considered possible, based on the description given in the natural language scenarios 110 and/or the trace-based specification 130. In reality, the behavior of the system is much more complex than considered in the natural language scenarios 110 and/or the trace specification 130 due to interactions between components, which users (and often developers) do not consider. As a result, other behavior patterns prove to be possible and revised formal specification 175 is easier to examine and highlights error conditions to be addressed.

[0029] Revised Formal specification 175 allows model checking that is performed to detect omissions and race conditions. Revised Formal specification 175 also allows verification of the requirements established by natural language scenarios 110. Revised formal specification 175 provides for the formal analysis of interactions between processes and the proofs of properties, such as the presence or absence of livelock or deadlock. A deadlock condition is defined as one where nothing

happens after an action is taken. A livelock condition is defined as one where unpredictable results occur after an action. These conditions can be fatal in the software development process. Many other behaviors, such as avoidance of particular conditions, or recovery from particular conditions, can also be proven.

[0030] Formal specification converter 180 is a converter for formal specification 175. Formal specification converter 180 takes the higher level formal specification 170 and converts it to a lower-level specification 185 (possibly written in a different formal specification language or notation) that is suitable for input into a code generator 190. Preferably, low level specification 185 is in the B specification language, or some other specification language or notation. Further, code generator 190, preferably, is (part of) a B specification language toolkit. Code generator 190 generates code 195.

[0031] Code 195 preferably comprises a sequence of instructions to be executed as a software system. Code 195 is an accurate description of the system defined by natural language scenarios 110 that is mathematically provable. Code 195 is used for verification 196 of the natural language scenarios 110 as well as trace specification 130. Further, code 195 may be used as a basis for model checking 197. Code 195 comprises executable computer code, which may be translated before execution. Translator 200 receives code 195. Translator 200 is a compiler, interpreter, translator, transformer, and the like that generates instructions at a low level, such as machine code.

[0032] Either at the point of generating code 195, or ideally at the time of generating a revised formal specification 175, errors and problems are likely to arise, especially during the

first iteration, as in all development situations. Incompatible scenarios are likely to raise the need for adjustments, improvements, and corrections. Natural language scenarios 110 may have to be corrected or changed. In these instances, software development system 100 allows for the modification of (or accepts the user's modifications to) the natural language scenarios 110 and re-executes the actions disclosed above. Moreover, if the requirements change, natural language scenarios 110 may be modified accordingly. Thus, natural language scenarios 110 are adaptable. Advanced users may wish to make amendments at the level of the traces specification 130 or the process-based specification 160 or even at the level of the revised formal specification 175. This action is permitted, but means that changes only take effect from that point on, and the mathematically provable equivalence to the natural language scenarios 110 is not maintained.

[0033] According to the disclosed embodiments of the present invention, software development system 100 provides mathematical traceability in that it allows for mathematical logic to demonstrate what was produced as code 195 matches what was defined by natural language scenarios 110. Thus, code 195 is provably correct with respect to natural language scenarios 110. Further, executable code 195 may be mechanically regenerated when requirements dictate a change in the high level specifications. Any change in the system defined by natural language scenarios 110 introduces the risk of new errors that necessitate re-testing and re-validation. This risk is reduced according to the disclosed embodiments.

[0034] The use of natural language scenarios 110 is effective in specifying a wide range of types of software. Automating the translation from natural language scenarios 110 to formal

specification 185 reduces the need for testing yet produces high quality software systems. The need for testing is important, especially in those classes of systems where all possible execution paths cannot be tested. Any reduction in this need is desirable. The benefits of formal specification and development provide assurance that the software is operating correctly. Due to the high assurance techniques the amount of testing is reduced and less time is spent on coding and developing test cases. The result is reduced development time, higher quality systems, and less expense.

[0035] The disclosed embodiments provide for full automatic code generation that is efficient, that reduces the opportunity for programming errors, and that supports the whole system development life cycle. The disclosed embodiments also fit with existing processes. Specifically, through formal methods, the disclosed embodiments augment an informal case-based approach based on scenarios, thus bringing the benefits of correctness, precision, concision, accuracy, proof, and automatic code generation. The disclosed embodiments accept scenario-level specifications as input and convert them into a mathematics-based formal specification language. From the formal specification language, model checking and proofs of correctness are undertaken.

[0036] Once the specification is checked and corrections are made (involving a re-iteration of the development described above), the formal specification is converted into executable computer code. The formal specification provides means to check the scenarios for potential errors that would be difficult or nearly impossible to detect if those specifications had no underlying formal mathematical foundation. Thus, an end-to-end tool is provided that allows the creation of the high level

specification for a software, or other, system, the conversion into a formal specification language, a verification of the formal specification for errors, and the production of executable computer code correctly representing the original high level specification. Thus, the disclosed invention realizes the long-sought goal in computer science of a tractable, automated means to obtain executable code that is provably equivalent to the application requirements (that is, the high level specification of the application).

[0037] In resolving conflicts between the behaviors described in natural language scenarios 110, an acceptable set of behaviors is determined and used in deriving the requirements specification of the system. Natural language scenarios 110 are expressed stating the actions that the system performs in response to various stimuli, external or internal. The actions and stimuli are chosen from a dictionary (representing the application domain context) of previously determined parameters for given processes and the proposed system.

[0038] In formal methods, a trace is the sequence of the events that a process is engaged in up to any given point in time. The set of traces of a process represents the set of all possible behavior patterns of that process. In many cases, the set of traces is an infinite set of finite sequences of events. The set of traces of a system is calculated from the sets of traces of its constituent processes, using the rules of a suitable formal specification language. In essence, a scenario is a constrained natural language description of some subset of the set of traces of the system.

[0039] Moreover, the use of the trace semantics is highly tractable. It is not necessary, however, to be able to list all of the traces of the system. Rather, the set of traces is given

in intention; that is, a formula for calculating the set of traces is given, and manipulated, again using rules given in the formal specification language. Given a sequence of events, it is possible to determine whether it denotes acceptable execution of the system by determining whether or not it is a valid member of the set of traces of that system.

[0040] Therefore, from natural language scenarios 110, using a dictionary of events and actions written in a constrained way embodying domain parameters 115, it is possible to go to trace specification 130 of the system behavior. The disclosed embodiments then devise a mechanism for extracting a more useful process-based formal specification from trace specification 130.

[0041] Figure 2 is a flowchart for deriving formal specifications and code from scenarios according to the disclosed embodiments. Figure 1, however, is not limited by the embodiments disclosed by Figure 2.

[0042] Step 202 executes by inputting scenarios written according to proposed system natural language text. Example 1 of the disclosed embodiments below, is a natural language scenario for a software system of an autonomous, agent-based ground system for satellite control.

[0043] Example 1 of the Disclosed Embodiments

[0044] If the Spacecraft Monitoring Agent receives a "fault" advisory from the spacecraft

[0045] The agent sends the fault to the Fault Resolution Agent

[0046] OR

[0047] If the Spacecraft Monitoring Agent receives engineering data from the spacecraft

[0048] The agent will send the data to the Trending Agent

[0049] The software system uses agents as surrogates for human spacecraft controllers. In Example 1, the natural language scenario specifies that, when the agent receives a spacecraft fault, it will then send it on to a fault resolution agent that will analyze it for possible actions to take. The natural language scenario also states that if the agent receives engineering data from the spacecraft, it will send it to a trending agent.

[0050] Step 204 executes by inputting domain parameters from a database. The database of parameters includes domain specific terms to help define the scenarios. Such databases are developed for specific domains and inputted into the software development process, such as software development system 100 of Figure 1.

[0051] Step 206 executes by generating traces for the natural language scenarios. Example 2 of the disclosed embodiments below discloses the equivalent structured text of a trace that the natural language scenario disclosed in Example 1 would be converted into.

[0052] Example 2 of the Disclosed Embodiments

[0053] *inSCMA?fault* from Spacecraft

[0054] then *outSCMA!* to FRA

[0055] else

[0056] *inengSCMA!data* from Spacecraft

[0057] then *outengSCMA!data* to TREND

[0058] Example 3 of the disclosed embodiments below discloses the traces of the specification derived from the structured text.

[0059] Example 3 of the Disclosed Embodiments

[0060] 1. $tSCMA \supseteq \{<>, < inSCMAfault, outSCMAfault >\}$

[0061] 2. $+\{<>, < inengSCMAdata > < inengSCMAdata, outSCMAdata >\}$

[0062] Step 208 executes by producing the process-based formal specification from the traces. Example 4 of the disclosed embodiments discloses the formal specification produced from the traces. Also, following Step 208, error checking and trouble-shooting may occur.

[0063] Example 4 of the Disclosed Embodiments

[0064] $SCMA = in\ SCMA?fault \rightarrow (outSCMA!fault \rightarrow STOP)$

[0065] $\mid (inengSCMA?data \rightarrow outengSCMA!data \rightarrow STOP)$

[0066] Step 210 executes by generating code from the lower level specification. The derived code is executable computer code that is a sequence of instructions in a suitable notation for execution within a software system. The generated code includes commands for the domain specified earlier. The derived code is an accurate description of the desired software system that exactly and mathematically provably corresponds to the natural language scenarios given as input in step 202. Step 212 executes by translating the code into machine code or other low level instructions for use in the subject satellite system.

[0067] The flowchart of Figure 2 allows for verification and validation between Step 202 and Step 212. As indicated on the flowchart, the disclosed process may be repeated at any time to

correct mistakes, change requirements, and the like. In other words, at any step in the disclosed process, the disclosed embodiments may repeat a step or a series of steps, or may be interrupted by the user, if an error is found, or if the user wishes simply to repeat the step, or steps, or to make a change that may not represent an error but rather a desired change or enhancement. Further, the process may be revisited at a later time to modify the software requirements or specifications. Thus, the disclosed embodiments provide flexibility over conventional systems by allowing testing and validation to occur, and to easily modify the system.

[0068] Further, according to the disclosed embodiments of the present invention, the disclosure of Figure 2 allows for the reverse process to be implemented. In other words, generated code may be "reverse engineered" to derive formal specifications, which may be analyzed more easily and often may be understood more easily. This approach may even be used to derive traces and natural language scenarios as a means of explaining (or paraphrasing) the operation of existing systems. This feature is desirable for legacy systems or those systems where the original specifications or requirements are not available. Once the natural language scenarios are derived, they can be modified to update, correct, or change the requirements of the software system. Further, the disclosed embodiments are not just applicable to software systems. The disclosed embodiments are applicable to any system, device, or process that uses instructions to perform an action. For example, the disclosed embodiments are applicable to systems that describe or generate instructions for systems that result in the manufacture of products or items.

[0069] It will be apparent to those skilled in the art that various modifications and variations can be made in the present invention without departing form the spirit or scope of the invention. Thus, it is intended that the present invention covers the modifications and variations of this invention provided that they come within the scope of any claims and their equivalents.